**TP 3-4**

**Programmation Objet Java**

## 1 Objectifs

L'objectif de ce TP est de consolider les notions de classe abstraite et d'héritage, d'une part et d'interface et de réalisation, d'autre part. Nous réviserons également la notion de paquetage. Pour cela, nous travaillerons dans le contexte déjà développé durant les séances précédentes.

## 2 Contexte

Notre travail sera centré sur le paquetage stock, celui-ci contiendra les différentes classes vues en petite classe : ***Stock*** (abstraite) et **StockTableau**, ainsi que les exceptions : **StockFull**, **StockEmpty**, **ProduitNull**. Nous allons étendre le contexte précédent :

Nous allons réaliser la classe **StockListe** (une autre concrétisation de la classe Stock (abstraite)). Cette réalisation n'utilisera pas un tableau pour stocker les produits, mais une **LinkedList <Produit>**. Ce sera pour nous l'occasion d'utiliser une classe du package **java.util** et la documentation qui lui est associée.

Les méthodes **toString**() des classes **StockListe** et **StockTableau** seront ensuite « factorisées » au niveau de ***Stock*** (abstraite). Pour cela, nous utiliserons la notion d'itérateur qui fournit le moyen de parcourir une collection indépendamment de sa structure.

Cette description correspond au diagramme UML de la figure suivante.

Les paquetages **date**, **produit** et **stock** vous sont fournis. Il contiennent les classes nécessaires au TP. Ils sont accessibles dans le répertoire **: ~lib-src/FIP/INF111/TP3-4**

La classe **StockListe** sera à écrire.

## 3 Compilation et Exécution avec des paquetages

Pour être dans le paquetage **produit**, une classe doit être dans le répertoire correspondant et contenir la déclaration : **package produit;**

Pour utiliser la classe **Date** du paquetage **date**, il faut soit:

- être dans le répertoire **date** contenant la classe **Date**
- utiliser systématiquement le nom **date.Date**
- importer le paquetage par l'instruction :
  - ▲ import **date.Date;** pour importer uniquement **Date**
  - ▲ import **date.*;** pour importer toutes les classes du paquetage **date**

Pour compiler ces classes, placez-vous dans le répertoire **TP3-4** (père de **date** et **produit**) et utilisez la commande **javac date/*.java produit/*.java**.

Pour lancer l'exécution (l'interprétation du bytecode) par la JVM du **main** de la classe **Produit**, il faut être dans le répertoire **TP3-4** et utiliser la commande

**java produit.Produit**.

Pour lancer l'exécution du **main** de la classe **TestTableau**, il faut être dans le répertoire **TP3-4**, compiler la classe **TestTableau**, puis utiliser la commande

**java stock.TestTableau**.

## 4 Exercice : écriture de StockListe

Réalisez la classe **StockListe** qui utilise une **LinkedList <Produit>** pour stocker les produits. Pour cela, ajouter au paquetage **stock**, une classe **StockListe** suivant la conception décrite dans le diagramme UML. Pour vous simplifier l'écriture de la classe **StockListe** recopiez le fichier **StockTableau.java** dans le fichier **StockListe.java**, puis modifiez le fichier **StockListe.java**.

Testez en utilisant la classe **TestListe**.

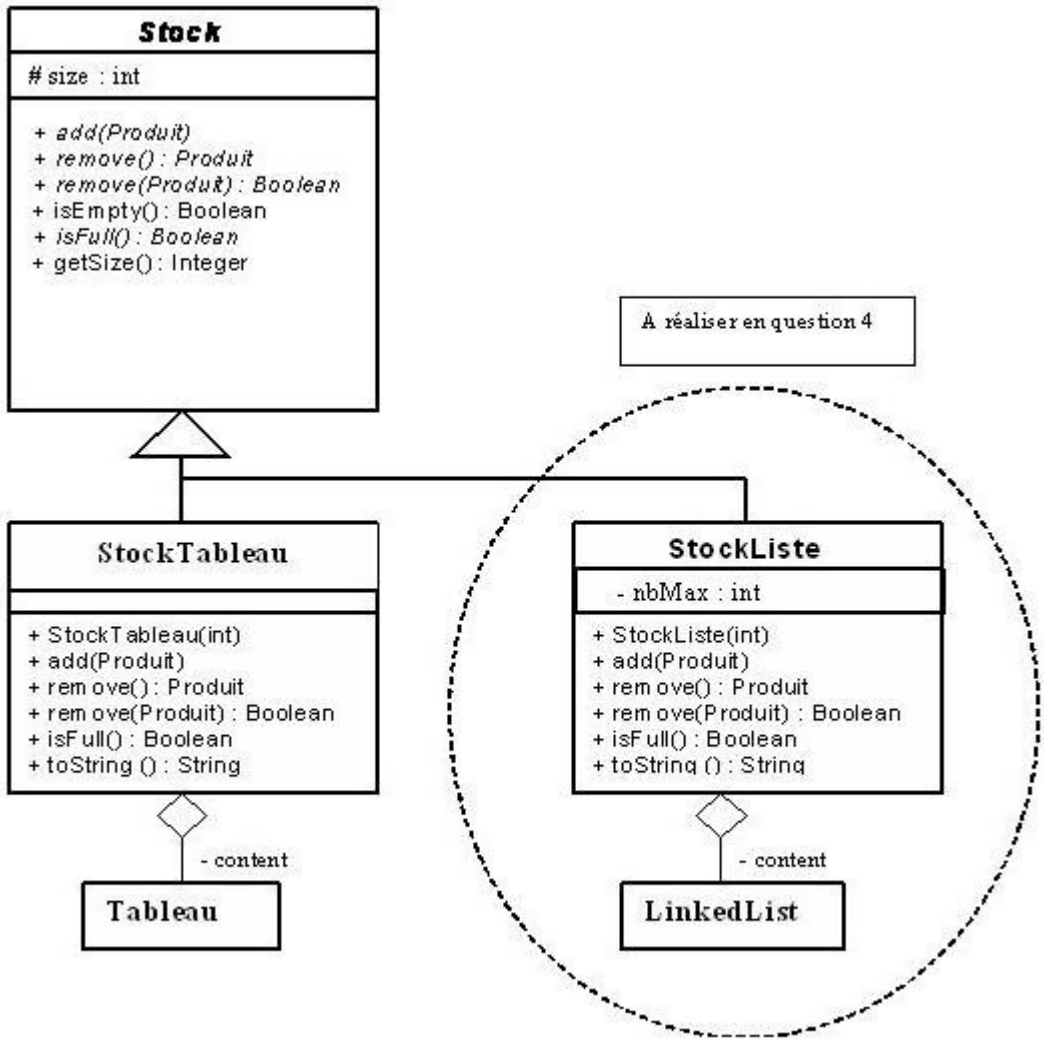


*Diagramme de classes à la fin de la question 4.*

## 5 Exercice : factorisation de toString()

Le code des méthodes **toString()** présentes dans les classes **StockTableau** et **StockListe** sont "quasiment" identiques. On souhaite factoriser au niveau de la classe abstraite **Stock** la réalisation de la méthode **toString()**. Pour cela on créera une méthode abstraite **iemeElement(int i):Produit** dans la classe **Stock**. La valeur retournée par **iemeElement** devra être le ième produit du Stock. La méthode **iemeElement** devra être concrétisée dans les classes **StockTableau** et **StockListe**. Remplacez la méthode **toString()** abstraite de la classe **Stock** par une méthode **toString()** concrète. (inspirez vous du code du **toString()** de la classe **StockListe** et utilisez **iemeElement**). Les méthodes **toString()** seront supprimées des classes **StockTableau** et **StockListe**.

Testez en utilisant la classe **TestTableau** ou la classe **TestListe**.

## 6 La notion d'itérateur

Afin de permettre le parcours de toutes structures de données de manière uniforme la notion d'itérateur a été définie. C'est une entité abstraite qui connaît « l'intérieur » de la structure et mémorise le parcours déjà réalisé. Ainsi, le client d'un itérateur peut parcourir tous les éléments. Cette notion est spécifiée en Java par :

une interface **Iterator <E>**

Il y a une implémentation de cette interface pour toutes les collections du paquetage **java.util**

Une classe implémentant l'interface **Iterator <E>** offre (principalement) deux méthodes : **hasNext** et **next**. La première renvoie un booléen qui indique si le parcours de la structure de données est achevé ou non. La seconde renvoie l'objet **E** suivant de la structure de données. Le code complet de l'interface est fourni en annexe.

## 7 Exercice

Modifier la classe abstraite **Stock** en :

- rajoutant une méthode abstraite **iterator** de spécification:
  **public abstract Iterator <Produit> iterator() ;**
- modifiant la méthode **toString()** ( elle utilisera un objet de type **Iterator <Produit>** renvoyé par la méthode abstraite **iterator** de cette même classe).

Complétez la classe **StockListe** et la classe **StockTableau** pour qu'elles implémentent la méthode **iterator**.

- Dans **StockListe** la méthode **iterator** utilisera la méthode **iterator** de **LinkedList** (voir documentation de **LinkedList**; il s'agit d'une méthode héritée).
- Dans **StockTableau** la méthode **iterator** renverra (provisoirement) **null**.

Testez en utilisant la classe **TestListe.**

## 8 Exercice : l"itérateur" associé à un StockTableau (réalisation par classe interne)

En Java, dans une classe, il est possible de définir une autre classe qui sera dite interne. La classe interne aura accès à tous les attributs des objets de la classe la contenant.

Complétez la classe **StockTableau** en :

- modifiant la méthode **iterator** qui renverra **new IterInterneTableau()**
- réalisant la classe interne **IterInterneTableau** qui implémentera l'interface **Iterator <Produit>**. Il vous faudra écrire les 3 méthodes que toute classe qui implémente l'interface **Iterator <Produit>** doit fournir. Vous utiliserez une des particularités d'une classe interne : l'accès aux attributs de la classe qui la contient. Ainsi dans **IterInterneTableau** on peut utiliser l'attribut **content**.

Testez en utilisant la classe **TestTableau.**

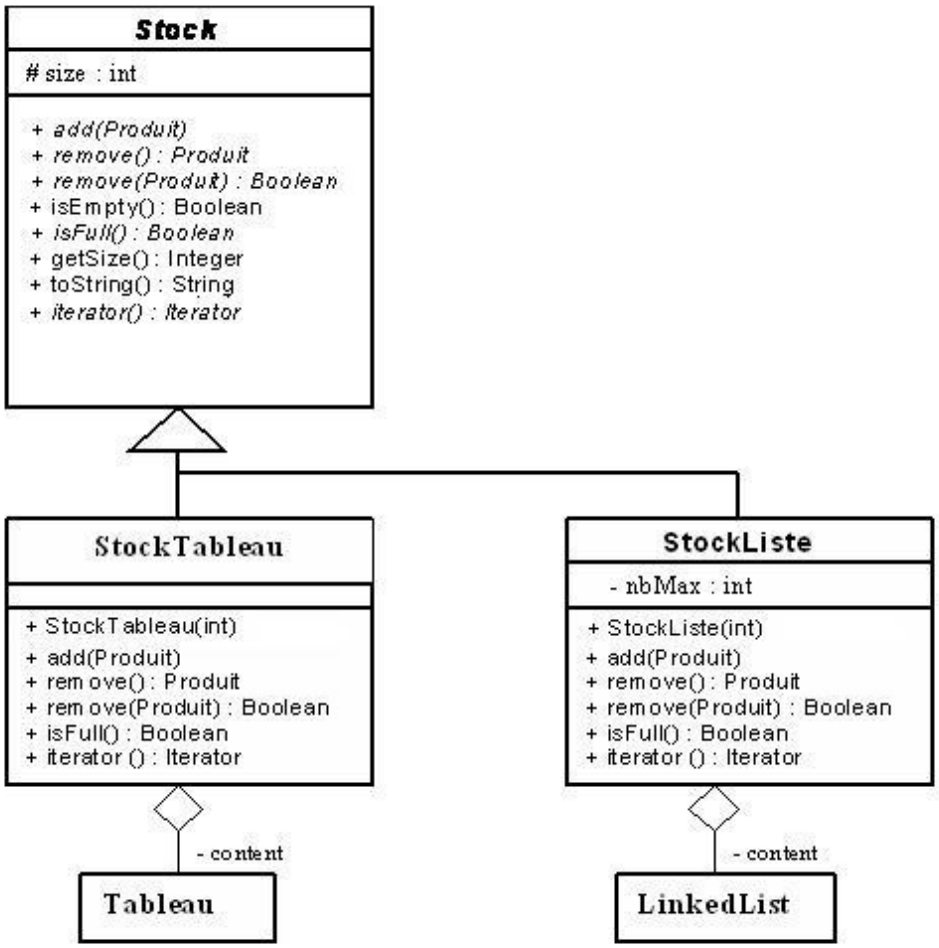


*Diagramme de classes à la fin de la question 8.*

## 9 Exercice : l"itérateur" associé à un StockTableau (réalisation par classe externe)

Externalisation de **IterInterneTableau :**

- modifiez la méthode **iterator** de la classe **StockTableau** qui renverra désormais **new IterExterneTableau( ???)**

- Réalisez dans le fichier **IterExterneTableau.java** du paquetage **stock** la classe **IterExterneTableau.** Vous pourrez utiliser un copier/coller de **IterInterneTableau**, mais ce ne sera pas suffisant car dans la classe **IterExterneTableau** on ne « connaît » ni l'attribut content, ni l'attribut size de **StockTableau**.

Testez en utilisant la classe **TestTableau.**

## 10 Annexes

```
public interface Iterator <E> {
    /**
    *    Returns  true  if the iteration has more elements. (In other
    *    words, returns  true   if  next   would return an element
    *    rather than throwing an exception.)
    *
    *    @return  true  if the iterator has more elements.
    */
    public boolean hasNext();

    /**
    *    Returns the next element in the iteration .
    *
    * @return the next element in the iteration .
    * @exception NoSuchElementException iteration has no more elements.
    */
    public <E>  next();

    /**
    *    Removes from the underlying collection the last element returned by the
    *    iterator ( optional operation ). This method can be called only once per
    *    call to  next . The behavior of an iterator is unspecified if
    *    the underlying collection is modified while the iteration is in
    *    progress in any way other than by calling this method.
    *
    *    @exception UnsupportedOperationException if the  remove
    *    operation is not supported by this Iterator .
    *    @exception IllegalStateException if the  next  method has not
    *    yet been called , or the  remove  method has already
    *    been called after the last call to the  next
    *    method.
    */
    public  void remove();
 }


public interface Iterable <T> {
    /**
    *    Returns an iterator over a set of elements of type T.
    */
    Iterator <T> iterator() ;

 }
```